

Tokens, Expressions and Control Structures

Key Concepts

Tokens|Keywords|Identifiers|Data types|User-defined types|Derived types|Symbolic constants|Declaration of variables|Initialization|Reference variables|Type compatibility|Scope resolution|Dereferencing|Memory management|Formatting the output|Type casting|Constructing expressions|Special assignment expressions|Implicit conversion|Operator overloading|Control structures

3.1

Introduction

As mentioned earlier, C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions. In this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.

3.2

Tokens

As we know, the **smallest individual units** in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically

similar to the C tokens with the exception of some additions and minor modifications.

3.3

Keywords

The keywords implement specific C++ language features. They are **explicitly reserved identifiers** and **cannot** be used as names for the program variables or other user-defined program elements.

[Table 3.1](#) gives the complete set of C++ keywords. Many of them are common to both C and C++. The ANSI C keywords are shown in boldface. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. ANSI C++ standards committee has added some more keywords to make the language more versatile. These are shown separately. Meaning and purpose of all C++ keywords are given in Appendix D.

Table 3.1 *C++ keywords*

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Note: The ANSI C keywords are shown in bold face.

3.4 Identifiers and Constants

Identifiers refer to the names of **variables, functions, arrays, classes,** etc., created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only **alphabetic characters, digits and underscores** are permitted.
- The name **cannot** start with a digit.
- Uppercase and lowercase letters **are distinct.**
- A **declared keyword** cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable which is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.

Constants refer to fixed values that **do not change** during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Examples:

```
123          // decimal integer
12.34        // floating point integer
037          // octal integer
0X2          // hexadecimal integer
"C++"        // string constant
'A'          // character constant
L'ab'        // wide-character constant
```

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

C++ also recognizes all the backslash character constants available in C.



NOTE: C++ supports two types of string representation — the C-style character string and the string class type introduced with Standard C++. Although the use of the string class type is recommended, it is advisable to understand and use C-style strings in some situations. The string class type strings support many features and are discussed in detail in Chapter 15.

Basic Data Types

Data types in C++ can be classified under various categories as shown in [Fig. 3.1](#).

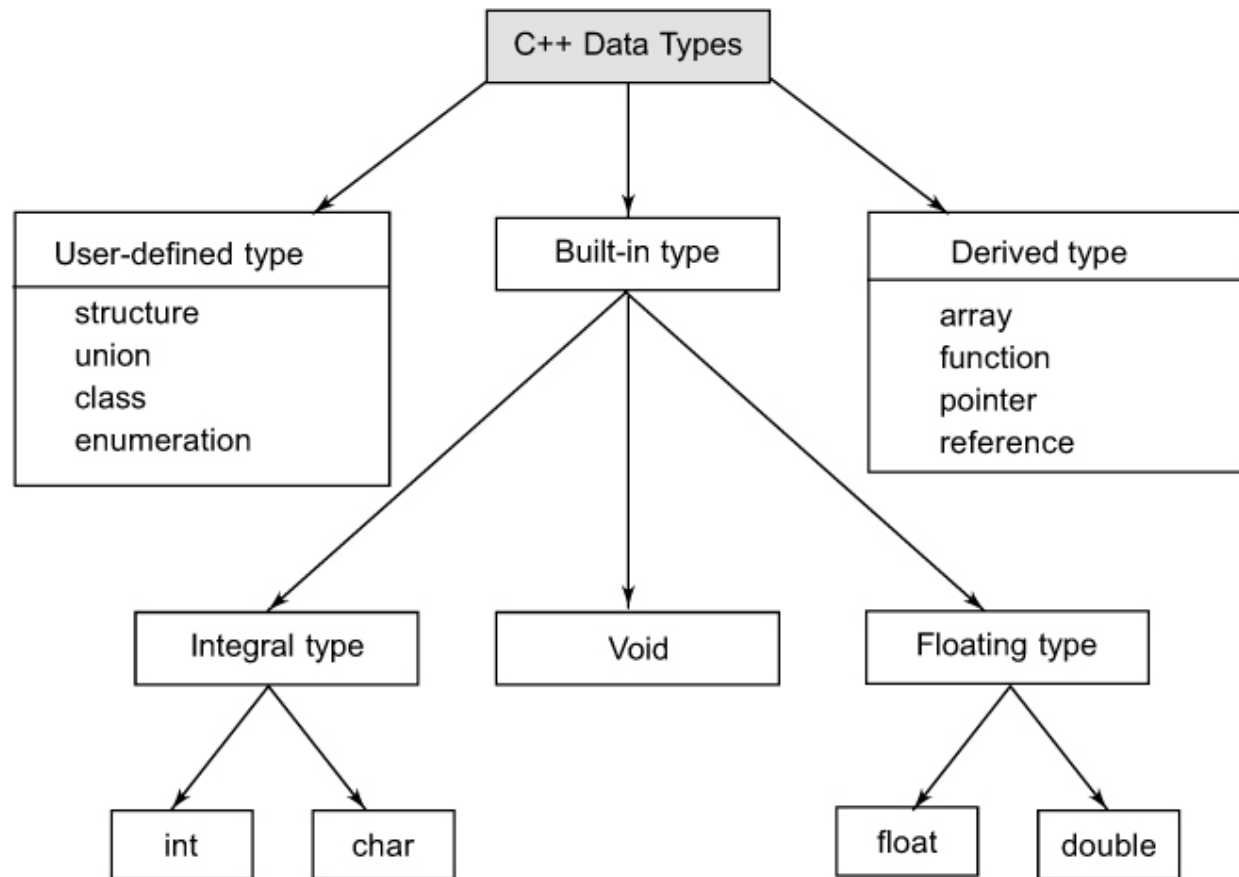


Fig. 3.1 *Hierarchy of C++ data types*

Both C and C++ compilers support all the built-in (also known as *basic* or *fundamental*) data types. With the exception of **void**, the basic data types may have several *modifiers* preceding them to serve the needs of various situations. The modifiers **signed**, **unsigned**, **long**, and **short** may be applied to character and integer basic data types. However, the modifier **long** may also be applied to **double**. Data type representation is machine specific in C++. [Table 3.2](#) lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

Table 3.2 *Size and range of C++ basic data types*

Type	Bytes	Range
char	1	−128 to 127
unsigned char	1	0 to 255
signed char	1	− 128 to 127
int	2	− 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	− 32768 to 32767
short int	2	− 32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	−32768 to 32767
long int	4	−2147483648 to 2147483647
signed long int	4	−2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E−38 to 3.4E+38
double	8	1.7E−308 to 1.7E+308
long double	10	3.4E−4932 to 1.1E+4932

ANSI C++ committee has added two more data types, **bool** and **wchar_t**. They are discussed in Chapter 16.

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void functl(void);
```

Another interesting use of **void** is in the declaration of generic pointers. Example:

```
void *gp;    // gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

```
int *ip;    // int pointer  
gp = ip;    // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to a **void** value.

Assigning any pointer type to a **void** pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a **void** pointer to a nonvoid pointer without using a cast to nonvoid pointer type. This is not allowed in C++. For example,

```
void *ptr1;  
char *ptr2;  
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *)ptr1;
```

3.6 User-Defined Data Types

Structures and Unions

Standalone variables of primitive types are not sufficient enough to handle real world problems. It is often required to **group logically related data items together**. While arrays are used to group together similar type data elements, structures are used for grouping together elements with dissimilar types.

The general format of a structure definition is as follows:


```
struct name
{
    data_type member1;
    data_type member2;
    . . . . .
    . . . . .
};
```

Let us take the example of a book, which has several attributes such as title, number of pages, price, etc. We can realize a book using structures as shown below:

```
struct book
{
    chartitle[25];
    charauthor[25];
    int pages;
    float price;
};
struct book book1, book2, book3;
```

Here book1, book2 and book3 are declared as variables of the user-defined type book. We can access the member elements of a structure by using the dot (.) operator, as shown below:

```
book1.pages=550;
book2.price=225.75;
```

Unions are conceptually similar to structures as they allow us to group together dissimilar type elements inside a single unit. But there are significant differences between structures and unions as far as their implementation is concerned. The size of a structure type is

equal to the sum of the sizes of individual member types. However, the size of a union is equal to the size of its largest member element. For instance, consider the following union declaration:

```
union result
{
    int marks;
    char grade;
    float percent;
};
```

The union result will occupy four bytes in memory as its largest size member element is the floating type variable percent. However, if we had defined result as a structure then it would have occupied seven bytes in memory that is, the sum of the sizes of individual member elements. Thus, in case of unions the same memory space is used for representing different member elements. As a result, union members can only be manipulated exclusive of each other. In simple words, we can say that unions are memory-efficient alternatives of structures particularly in situations where it is not required to access the different member elements simultaneously.

In C++, structures and unions can be used just like they are used in C. However, there is an interesting side to C++ structures that we will study in Chapter 5.

Table 3.3 *Difference between structures and unions*

<i>Structure</i>	<i>Union</i>
A structure is defined with 'struct' keyword.	A union is defined with 'union' keyword.
All members of a structure can be manipulated simultaneously.	The members of a union can be manipulated only one at a time.
The size of a structure object is equal to the sum of the	The size of a union object is equal to the size of largest member

individual sizes of the member objects.	object.
Structure members are allocated distinct memory locations.	Union members share common memory space for their exclusive usage.
Structures are not considered as memory efficient in comparison to unions.	Unions are considered as memory efficient particularly in situations when the members are not required to be accessed simultaneously.
Structures in C++ behave just like a class . Almost everything that can be achieved with a class can also be done with structures.	Unions retain their core functionality in C++ with slight add-ons like declaration of anonymous unions.

Classes

C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming. Other details about classes are discussed in Chapter 5.

Enumerated Data Type

An enumerated data type is another **user-defined type** which provides a way for attaching **names** to **numbers**, thereby increasing comprehensibility of the code. The **enum** keyword (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
```

```
enum position{off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names **shape**, **colour**, and **position** become new type names. By using these tag names, we can declare new variables. Examples:

```
shape ellipse;    // ellipse is of type shape
colour background; // background is of type colour
```

ANSI C defines the types of **enums** to be **ints**. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an **int** value to be automatically converted to an **enum** value. Examples:

```
colour background = blue;    // allowed
colour background = 7;       // Error in C++
colour background = (colour) 7; // OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c = red; // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by **explicitly assigning integer values** to the enumerators. For example,

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous **enums** (i.e., **enums** without tag names). Example:

```
enum{off, on};
```

Here, **off** is 0 and **on** is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;  
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement. Example:

```
enum shape  
{  
    circle,  
    rectangle,  
    triangle  
};  
int main()  
{  
    cout << "Enter shape code:";  
    int code;  
    cin >> code;  
    while(code >= circle && code <= triangle)  
    {  
  
        switch(code)  
        {  
            case circle:  
                .....  
                .....  
                break;  
            case rectangle:  
                .....  
                .....  
                break;  
            case triangle:  
                .....  
                .....  
                break;  
        }  
    }  
}
```

```

    }
    cout << "Enter shape code:";
    cin >> code;
}
cout << "BYE \n";

return 0;
}

```

ANSI C permits an **enum** to be defined within a structure or a class, but the **enum** is globally visible. In C++, an **enum** defined within a class (or structure) is local to that class (or structure) only.

3.7

Storage Classes

In addition to the data type, a variable, also has a **storage class** associated with it. The **storage class** of a variable specifies the **lifetime** and **visibility** of a variable **within** the program. **Lifetime** refers to the longevity of a variable or the duration till which a variable remains active during program execution. **Visibility**, on the other hand signifies the scope of a variable i.e., in which program modules the variable is accessible. There are four types of storage classes, as explained below:

Automatic: It is the **default storage class** of any type of variable. Its **visibility** is restricted to the **function in which it is declared**. Further, its **lifetime** is also limited till the time its container function is **executing**. That is, it is created as soon as its declaration statement is encountered and is destroyed as soon the program control leaves its **container function block**.

External: As the name suggests, an external variable is **declared outside of a function** but is **accessible inside the function block**. Also

called **global variable**, its **visibility** is spread all across the program that means, it is **accessible by all the functions** present in the program. The **lifetime** of an external variable is same as the lifetime of a program.

Static: A static variable has the **visibility of a local variable** but the **lifetime of an external variable**. That means, once declared inside a function block, **it does not get destroyed** after the function is executed, but **retains its value** so that it can be used by future function calls.

Register: Similar in behaviour to an automatic variable, a register variable differs in the **manner in which it is stored** in the memory. Unlike, automatic variables that are stored in the **primary memory**, the register variables are stored in **CPU registers**. The objective of storing a variable in registers is to **increase its access speed**, which eventually makes the program run faster. However, it may be noted that it is not an **obligation** for a compiler to store a register type variable **only in CPU registers**, but if there are no registers vacant to accommodate the variable then it is stored just like any other automatic variable.

Table 3.4 gives a summary of the four storage classes:

Table 3.4 **Storage Classes**

	<i>Automatic</i>	<i>External</i>	<i>Static</i>	<i>Register</i>
Lifetime	Function block	Entire program	Entire program	Function block
Visibility	Local	Global	Local	Local
Initial Value	Garbage	0	0	Garbage
Storage	Stack segment	Data segment	Data segment	CPU Registers
Purpose	Local variables used by a single function	Global variables used throughout the program	Local variables retaining their values throughout the program	Variables using CPU registers for storage purpose
Keyword	auto	extern	static	Register

Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
char string[3] = "xyz";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character `\0` in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string[4] = "xyz"; // O.K. for C++
```

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ program more reliable and readable. All the features of C++ functions are discussed in Chapter 4.

Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip;      // int pointer  
ip = &x;      // address of x assigned to ip  
*ip = 10;     // 10 assigned to x through indirection
```


C++ adds the concept of constant pointer and pointer to a constant.

```
char * const ptr1 = "GOOD";    // constant pointer
```

We cannot modify the address that **ptr1** is initialized to.

```
int const * ptr2 = &m; // pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

```
const char * const cp = "xyz";
```

This statement declares cp as a constant pointer to the string which has been declared a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

Pointers are extensively used in C++ for memory management and achieving polymorphism.

3.9

Symbolic Constants

There are two ways of creating symbolic constants in C++:

- Using the **qualifier const**, and
- Defining a **set of integer constants** using **enum** keyword.

In both C and C++, any value declared as **const** cannot be modified by the program in any way. However, there are some differences in implementation. In C++, we can use **const** in a constant expression, such as

```
const int size = 10;  
char name[size];
```

This would be illegal in C. **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

As with **long** and **short**, if we use the **const** modifier alone, it defaults to **int**. For example,

```
const size = 10;
```

means

```
const int size = 10;
```

The *named constants* are just like variables except that their values cannot be changed.

C++ requires a **const** to be initialized. ANSI C does not require an initializer; if none is given, it initializes the **const** to 0.

The scoping of **const** value differs. A **const** in C++ defaults to the internal linkage and therefore, it is local to the file where it is declared. In ANSI C, **const** values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as **static**. To give a **const** value an external linkage so that it can be referenced from another file, we must explicitly define it as an **extern** in C + + . Example :

```
extern const total = 100;
```

Another method of naming integer constants is by enumeration as under;

```
enum {X,Y,Z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const X = 0;  
const Y = 1;  
const Z = 2;
```

We can also assign values to X, Y, and Z explicitly. Example:

```
enum{X=100, Y=50, Z=200};
```

Such values can be any integer values. Enumerated data type has been discussed in detail in Section 3.6.

3.10

Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines **int**, **short int**, and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char**, **char**, and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way **char** constants are stored. In C, they are stored as **ints**, and therefore,

```
sizeof ( 'x' )
```

is equivalent to

```
sizeof(int)
```

in C. In C++, however, **char** is not promoted to the size of **int** and therefore,

```
sizeof('x')
```

equals

sizeof(char)

3.11 Declaration of Variables

We know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

The example below illustrates this point.

```
int main()
{
    float x;                // declaration
    float sum = 0;

    for(int i=1; i<5;       // declaration
    {
        cin >> x;
        sum = sum +x;
```

```

    }
    float average;           // declaration
    average = sum/(i-1);
    cout << average;

    return 0;
}

```

The only disadvantage of this style of declaration is that we cannot see all the variables used in a scope at a glance.

3.12 Dynamic Initialization of Variables

In C, a variable must be initialized using a **constant expression**, and the C compiler would fix the initialization code at the **time of compilation**. C++, however, **permits** initialization of the variables at **run time**. This is referred to as *dynamic initialization*. In C++, a variable can be initialized at run time **using expressions** at the place of **declaration**. For example, the following are valid initialization statements:

```

.....
.....
int n = strlen(string);
.....
float area = 3.14159 * rad * rad;

```

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the example of the previous section

```

float average; // declare where it is necessary
average = sum/i;

```

can be combined into a single statement:

```
float average = sum/i; // initialize dynamically at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

3.13

Reference Variables

C++ introduces a new kind of variable known as the *reference* variable. A reference variable provides an *alias* (alternative name) for a previously defined variable. For example, if we make the variable **sum** a reference to the variable **total**, then **sum** and **total** can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data-type & reference-name = variable-name
```

Example:

```
float total = 100;  
float & sum = total;
```

total is a **float** type variable that has already been declared; **sum** is the **alternative name** declared to **represent** the variable **total**. Both the variables refer to the same data object in the memory. Now, the statements

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both `total` and `sum` to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol **&**. Here, **&** is not an address operator. The notation **float &** means reference to **float**. Other examples are:

```
int n[10];  
int &x = n[10];    // x is alias for n[10]  
char &a = '\n';    // initialize reference to a literal
```

The variable **x** is an alternative to the array element **n[10]**. The variable **a** is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant **\n** is stored.

The following references are also allowed:

```
i. int x;  
   int *p = &x;  
   int &m = *p;  
ii. int &n = 50;
```

The first set of declarations causes **m** to refer to **x** which is pointed to by the pointer **p** and the statement in (ii) creates an **int** object with value 50 and name **n**.

A major application of reference variables is in passing arguments to functions. Consider the following:

```
→ void f(int & x)           // uses reference
{
    x = x+10;               // x is incremented; so also m
}
int main()
{
    int m = 10;
    f(m);                   // function call
}
```

```
.....
.....
}
```

When the function call **f(m)** is executed, the following initialization occurs:

```
int & x = m;
```

Thus, x becomes an **alias** of m after executing the statement

```
f(m);
```

Such function calls are known as *call by reference*. This implementation is illustrated in [Fig. 3.2](#). Since the variables **x** and **m** are aliases, when the function increments **x**, **m** is also incremented. The value of **m** becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

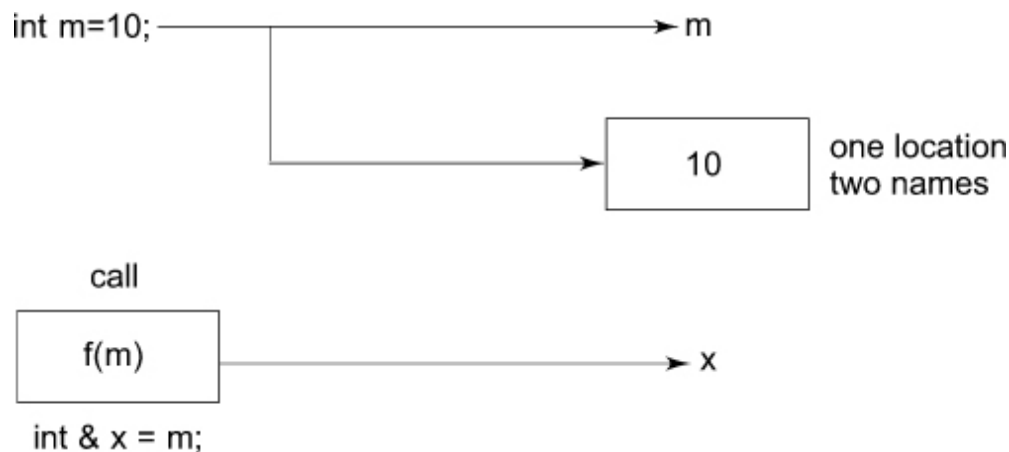


Fig. 3.2 *Call by reference mechanism*

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth. It is also important to note that references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

3.14

Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator `<<`, and the extraction operator `>>`. Other new operators are:

- `::` Scope resolution operator
- `::*` Pointer-to-member declarator
- `->*` Pointer-to-member operator
- `.*` Pointer-to-member operator
- delete** Memory release operator
- endl** Line feed operator
- new** Memory allocation operator
- setw** Field width operator

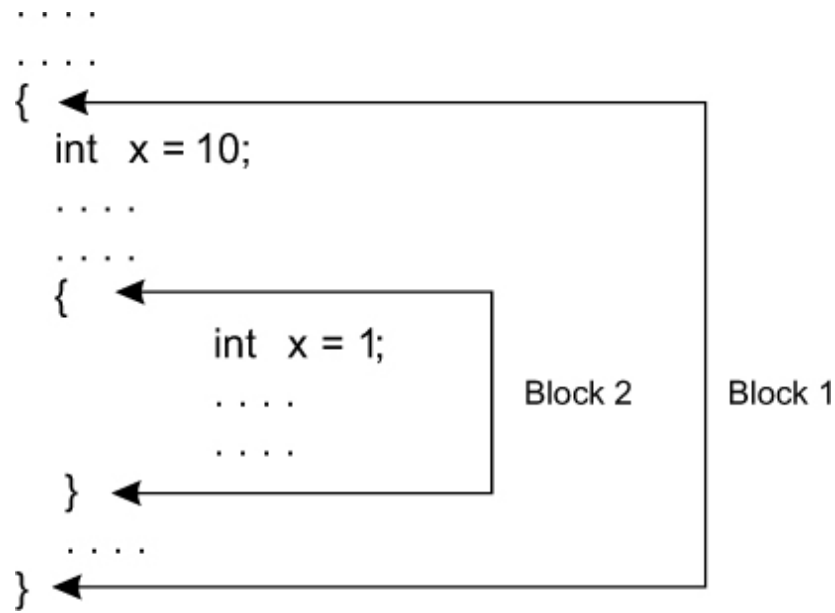
In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

3.15 Scope Resolution Operator

Like C, C++ is also a **block-structured language**. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program:

```
.....  
.....  
{  
    int x = 10;  
    .....  
    .....  
}  
.....  
.....  
{  
    int x = 1;  
    .....  
    .....  
}
```

The two declarations of x refer to **two different memory** locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common:



Block2 is contained in block1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of **x** causes it to refer to a different data object. Within the inner block, the variable **x** will refer to the data object declared therein.

In C, the **global version** of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator called the **scope resolution operator**. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, `::count` means the global version of the variable `count` (and not the local variable `count` declared in that block). Program 3.1 illustrates this feature.

Program 3.1 Scope Resolution Operator

```

#include <iostream>

using namespace std;

int m = 10;                // global m

int main()
{
    int m = 20;            // m redeclared, local to main

    {
        int k = m;
        int m = 30;        // m declared again
                           // local to inner block
        cout << "we are in inner block \n";
        cout << "k = " << k << "\n";
        cout << "m = " << m << "\n";
        cout << "::m = " << ::m << "\n";

    }

    cout << "\nWe are in outer block \n";
    cout << "m = " << m << "\n";
    cout << "::m = " << ::m << "\n";

    return 0;

}

```

The output of Program 3.1 would be:

```

We are in inner block
k = 20
m = 30 ::m = 10

```

We are in outer block

```
m = 20
::m = 10
```

In the above program, the variable **m** is declared at three places, namely, outside the **main()** function, inside the **main()**, and inside the inner block.



NOTE: It is to be noted **::m** will always refer to the global **m**. In the inner block, **::m** refers to the value 10 and not 20.

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs. This will be dealt in detail later when the classes are introduced.

3.16 Member Dereferencing Operators

As you know, C++ permits us to define a class containing various types of **data and functions** as members. C++ also permits us to **access the class members through pointers**. In order to achieve this, C++ provides a set of **three pointer-to-member operators**. [Table 3.5](#) shows these operators and their functions.

Table 3.5 *Member dereferencing operators*

Operator	Function
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, and therefore we defer the use of member dereferencing operators until then.

3.17 Memory Management Operators

C uses **malloc()** and **calloc()** functions to allocate memory dynamically at run time. Similarly, it uses the function **free()** to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as *free store* operators.

An object can be created by using **new**, and destroyed by using **delete**, as and when required. A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**. Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
pointer-variable = new data-type;
```

Here, *pointer-variable* is a pointer of type *data-type*. The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object. The *data-type* may be any valid data type. The *pointer-variable* holds the address of the memory space allocated.

Examples:

```
p = new int;  
q = new float;
```

where **p** is a pointer of type **int** and **q** is a pointer of type **float**. Here, **p** and **q** must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;  
float *q = new float;
```

Subsequently, the statements

```
*p = 25;  
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

We can also initialize the memory using the **new** operator. This is done as follows:

```
pointer-variable = new data-type(value);
```

Here, value specifies the initial value. Examples:

```
int *p = new int(25);  
float *q = new float(7.5);
```

As mentioned earlier, **new** can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

```
pointer-variable = new data-type[size];
```

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

When creating multi-dimensional arrays with **new**, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4];    // legal
array_ptr = new int[m][5][4];    // legal
array_ptr = new int[3][5][ ];    // illegal
array_ptr = new int[ ][5][4];    // illegal
```

The first dimension may be a variable whose value is supplied at runtime. All others must be constants.

Similar to the basic data type objects, the **new** operator can also be used for dynamically creating class objects. The size of the class object is automatically determined and the memory space is allocated accordingly. The general form of allocating class objects using **new** operator is:

```
class sample
{
    private:
        data-type d1;
        data-type d2;
        .
        .
    public:
        .
        .
};
main()
```



```
{  
    .  
    .  
    sample *ptr = new sample;  
    .  
}
```

Here, the size of the sample class object will be automatically determined and the corresponding memory space reserved by the **new** operator.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer-variable;
```

The *pointer-variable* is the pointer that points to a data object created with **new**. Examples:

```
delete p;  
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of **delete**:

```
delete [size] pointer-variable;
```

The *size* specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ]p;
```

will delete the entire array pointed to by **p**.

Program 3.2 demonstrates the use of new and delete operators for dynamic allocation and deallocation of memory.

Program 3.2 Use of new and delete Operators

```
#include <iostream>
#include <conio.h>

using namespace std;

void main()
{
    int *arr;
    int size;

    cout<<"Enter the size of the integer array: ";
    cin>>size;

    cout<<"Creating an array of size "<<size<<"..";
    arr = new int[size];

    cout<<"\nDynamic allocation of memory for array arr is
    successful.";

    delete arr;
    getch();
}
```

The output of Program 3.2 would be:

Enter the size of the integer array: 5

Creating an array of size 5..

Dynamic allocation of memory for array arr is successful.

What happens if sufficient memory is not available for allocation? In such cases, like **malloc** C, **new** returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows:

```
.....  
.....  
p = new int;  
if(!p)  
{  
    cout >> "allocation failed \n";  
}  
.....  
.....
```

If you are using Standard C++, then instead of null pointer the **bad_alloc** exception will be thrown. It is a system generated erroneous condition indicating memory allocation failure. To handle this situation, adequate exception handling code must be included in the program. We learn more about exception handling in Chapter 13.

Program 3.3 illustrates the use of **new** and **delete** operators for dynamic allocation of class objects. The program uses **bad_alloc** exception instead of null pointer for handling allocation failure.

Program 3.3 Use of bad_alloc Exception

```
#include <iostream>  
#include <conio.h>  
  
using namespace std;
```

```

class sample
{
    private:
        int data1;
        char data2;
    public:
        void set (int i,char c)
        {
            data1=i;
            data2=c;
        }
        void display(void)
        {
            cout<<"Data1 = "<<data1;
            cout<<"\nData2 = "<<data2;
        }
};

int main()
{
    sample *s_ptr;

    try
    {
        s_ptr = new sample;
    }

    catch (bad_alloc ba)
    {
        cout<<"Bad Allocation occurred...the program will terminate
        now..";
        return 1;
    }

    s_ptr->set(25,'A');
    s_ptr->display();
}

```

```
delete s_ptr;  
getch();  
}
```

The **new** operator offers the following advantages over the function **malloc()**.

1. It **automatically** computes the size of the data object. We need not use the operator **sizeof**.
2. It **automatically** returns the **correct pointer type**, so that there is no need to use a type cast.
3. It is possible to **initialize** the object while creating the **memory space**.
4. Like any other operator, **new** and **delete** can be **overloaded**.

3.18

Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character “\n”. For example, the statement

```
.....  
.....  
cout << m = << m << endl  
    << n = << n << endl  
    << p = << p << endl;  
.....  
.....
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:

m =

2	5	9	7
---	---	---	---

n =

1	4
---	---

p =

1	7	5
---	---	---

It is important to note that this form is not the ideal output. It should rather appear as under:

m = 2597
n = 14
p = 175

Here, the numbers are *right-justified*. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout << setw(5) << sum << endl;
```

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

		3	4	5
--	--	---	---	---

Program 3.4 Use of Manipulators

```

#include <iostream>
#include <iomanip>    // for setw

using namespace std;

int main()
{
    int Basic = 950, Allowance = 95, Total = 1045;

    cout << setw(10) << "Basic" << setw(10) << Basic <<
endl
        << setw(10) << "Allowance" << setw(10) <<
Allowance << endl
        << setw(10) << "Total" << setw(10) << Total << endl;
    return 0;
}

```

The output of Program 3.4 would be:

Basic	950
Allowance	95
Total	1045



NOTE: *Character strings are also printed right-justified.*

We may also control the precision of floating point numbers appearing in the output by using the `fixed` and `setprecision()` manipulators, as shown below:

```
cout<<fixed<<setprecision(2)<<average;
```

The above statement will display the value of average variable on the screen restricted to a precision of two.

We can also write our own manipulators as follows:

```
#include <iostream>
ostream & symbol(ostream & output)
{
    return output << "\tRs";
}
```

The **symbol** is the new manipulator which represents **Rs**. The identifier **symbol** can be used whenever we need to display the string **Rs**.

3.19

Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression // C notation
type-name (expression) // C++ notation
```

Examples:

```
average = sum/(float)i; // C notation
average = sum/float(i); // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.


```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;  
p = int_pt(q);
```

Program 3.5 Explicit Type Casting

```
#include <iostream>  
#include <conio.h>  
using namespace std;  
  
int main()  
{  
    int intvar=25;  
    float floatvar=35.87;  
  
    cout<<"intvar = "<<intvar;  
    cout<<"\nfloatvar = "<<floatvar;  
    cout<<"\nfloat(intvar) = "<<float(intvar);  
    cout<<"\nint(floatvar) = "<<int(floatvar);  
  
    getch();  
}
```

The output of Program 3.5 would be:

```
intvar = 25  
floatvar = 35.87  
float(intvar) = 25  
int(floatvar) = 35
```

ANSI C++ adds the following new cast operators:

- `const_cast`
- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`

Application of these operators is discussed in Chapter 16.

3.20 Expressions and Their Types

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as *compound expressions*.

Constant Expressions

Constant Expressions consist of only constant values. Examples:

```
15  
20 + 5 / 2.0  
'x'
```

Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

Examples:

```
m  
m * n - 5  
  
m * 'x'  
5 + int(2.0)
```

where **m** and **n** are integer variables.

Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results. Examples:

```
x + y  
x * y / 10  
5 + float(10)  
10.75
```

where **x** and **y** are floating-point variables.

Pointer Expressions

Pointer Expressions produce address values. Examples:

```
m  
ptr  
ptr + 1  
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

Relational Expressions

Relational Expressions yield results of type **bool** which takes a value **true** or **false**. Examples:

```
x <= y  
a+b == c+d  
m+n > 100
```

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as *Boolean expressions*.

Logical Expressions

Logical Expressions combine two or more relational expressions and produces **bool** type results. Examples:

```
a > b  
x == 10 && x == 10 || y == 5
```

Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3    // Shift three bit position to left  
y >> 1    // Shift one bit position to right
```

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what are termed as *operator keywords* that can be used as alternative representation for operator symbols. Operator keywords are given in Chapter 16.

3.21 Special Assignment Expressions

Chained Assignment

```
x = (y = 10);  
or  
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a = b = 12.34;    // wrong
```

is illegal. This may be written as

```
float a=12.34, b=12.34    // correct
```

Embedded Assignment

```
x = (y = 50) + 10;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10 = 60 is assigned to x. This statement is identical to

```
y = 50;  
x = y + 10;
```

Compound Assignment

Like C, C++ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

```
variable1 = variable1 op variable2;
```

3.22

Implicit Conversions

We can mix data types in expressions. For example,

```
m = 5+2.75;
```

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as *implicit* or *automatic conversion*.

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the “smaller” type is converted to the “wider” type. For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int**. The “water-fall” model shown in [Fig. 3.3](#) illustrates this rule.

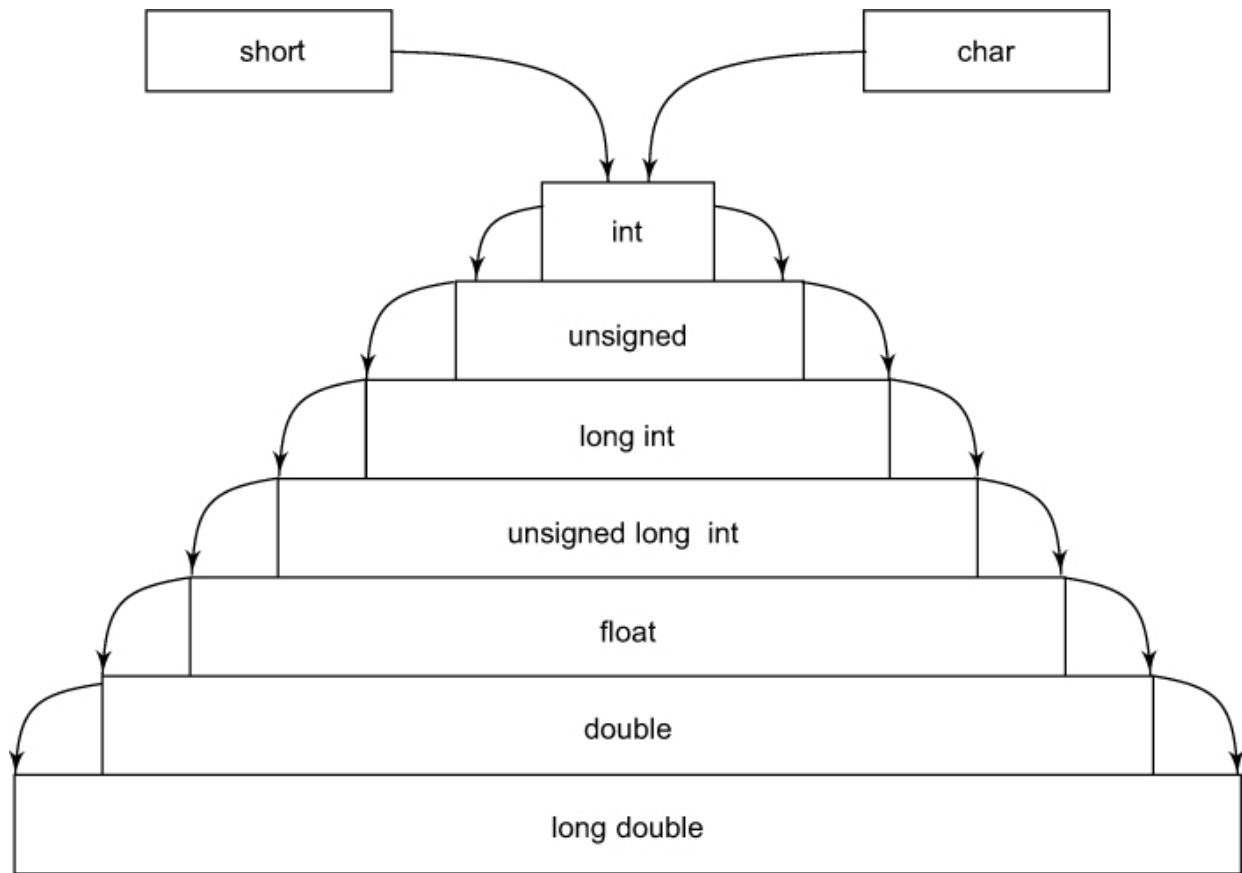


Fig. 3.3 *Water-fall model of type conversion*

Whenever a **char** or **short int** appears in an expression, it is converted to an **int**. This is called *integral widening conversion*. The implicit conversion is applied only after completing all integral widening conversions.

Table 3.6 *Results of Mixed-mode Operations*

<i>LHO</i> \ <i>RHO</i>	<i>char</i>	<i>short</i>	<i>int</i>	<i>long</i>	<i>float</i>	<i>double</i>	<i>long double</i>
<i>char</i>	int	int	int	long	float	double	long double
<i>short</i>	int	int	int	long	float	double	long double
<i>int</i>	int	int	int	long	float	double	long double
<i>long</i>	long	long	long	long	float	double	long double
<i>float</i>	float	float	float	float	float	double	long double
<i>double</i>	double	double	double	double	double	double	long double
<i>long double</i>	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand

LHO – Left-hand operand

3.23

Operator Overloading

As stated earlier, overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. Actually, we have used the concept of overloading in C also. For example, the operator `*` when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers. The number and type of operands decide the nature of operation to follow.

The input/output operators `<<` and `>>` are good examples of operator overloading. Although the built-in definition of the `<<` operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header file *iostream* where a number of overloading definitions for `<<` are included. Thus, the statement

```
cout << 75.86;
```

invokes the definition for displaying a **double** type value, and

```
cout << "well done";
```


invokes the definition for displaying a **char** value. However, none of these definitions in *iostream* affect the built-in meaning of the operator.

Similarly, we can define additional meanings to other C++ operators. For example, we can define + operator to add two structures or objects. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (. and .*), conditional operator (? :), scope resolution operator (::) and the size operator (**sizeof**). Definitions for operator overloading are discussed in detail in Chapter 7.

3.24

Operator Precedence

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. [Table 3.7](#) gives the precedence and associativity of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels *prefix* and *postfix* distinguish the uses of ++ and --. Also, the symbols +, -, *, and & are used as both unary and binary operators.

A complete list of ANSI C++ operators and their meanings, precedence, associativity and use are given in Appendix E.

Table 3.7 *Operator precedence and associativity*

Operator	Associativity
::	left to right
→ . () [] postfix ++ postfix --	left to right
prefix ++ prefix -- ~ ! unary + unary - unary *	
unary & (type) sizeof new delete	right to left
→ * *	left to right
* / %	left to right
+ -	left to right
<< >>	left to right
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^= =	left to right
, (comma)	

Note: The unary operations assume higher precedence.

3.25

Control Structures

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple to comprehend, while others are not. Experience has also shown that the number of bugs that occur is related to the format of the program. The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method

of achieving the objective of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures:

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition)

Figure 3.4 shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.

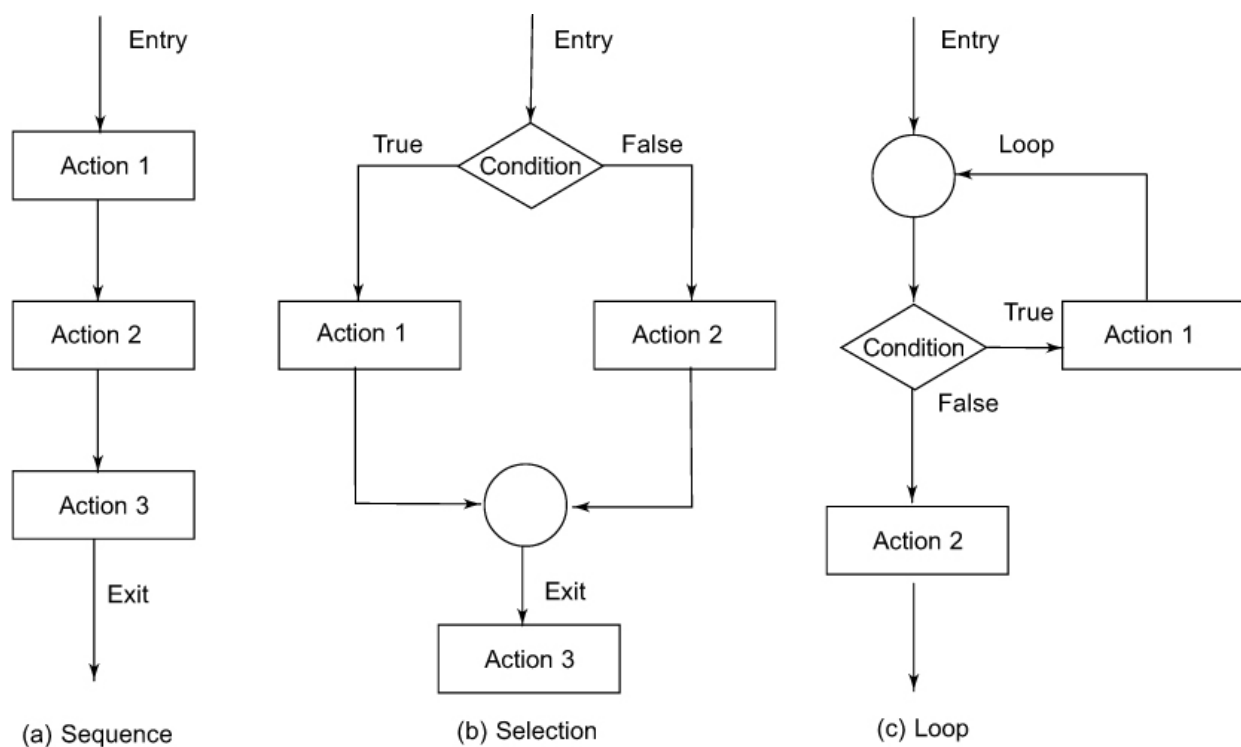
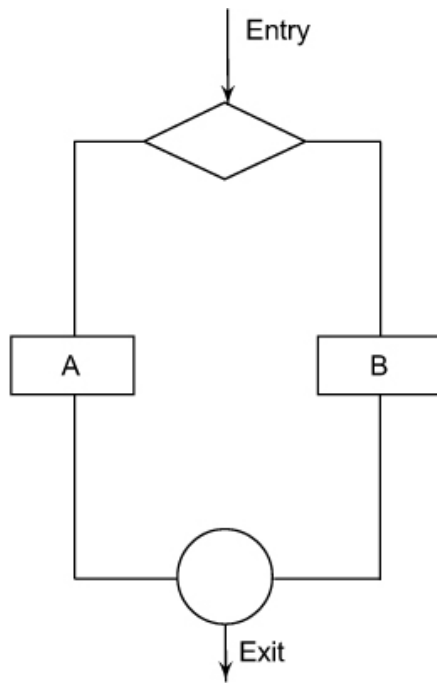


Fig. 3.4 Basic control structures

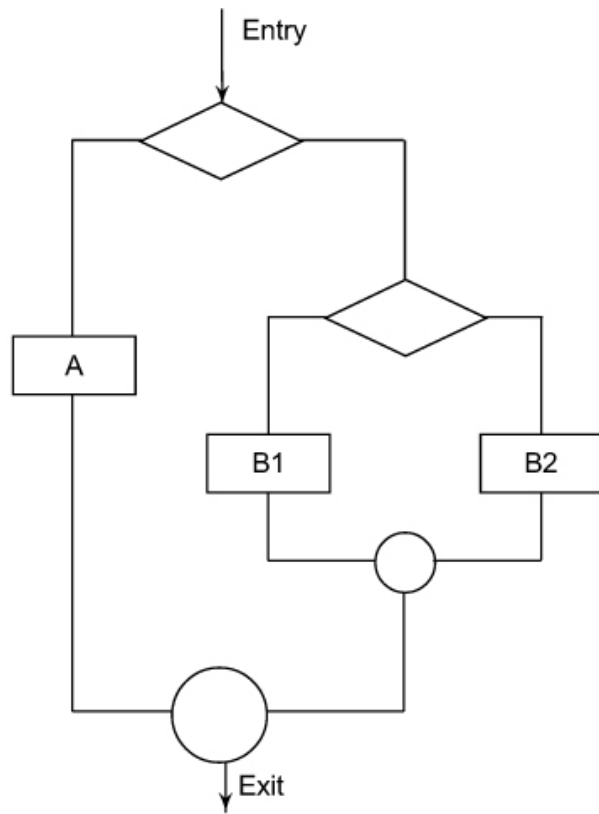
It is important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in [Figs. 3.5\(a\)](#), (b) and (c).

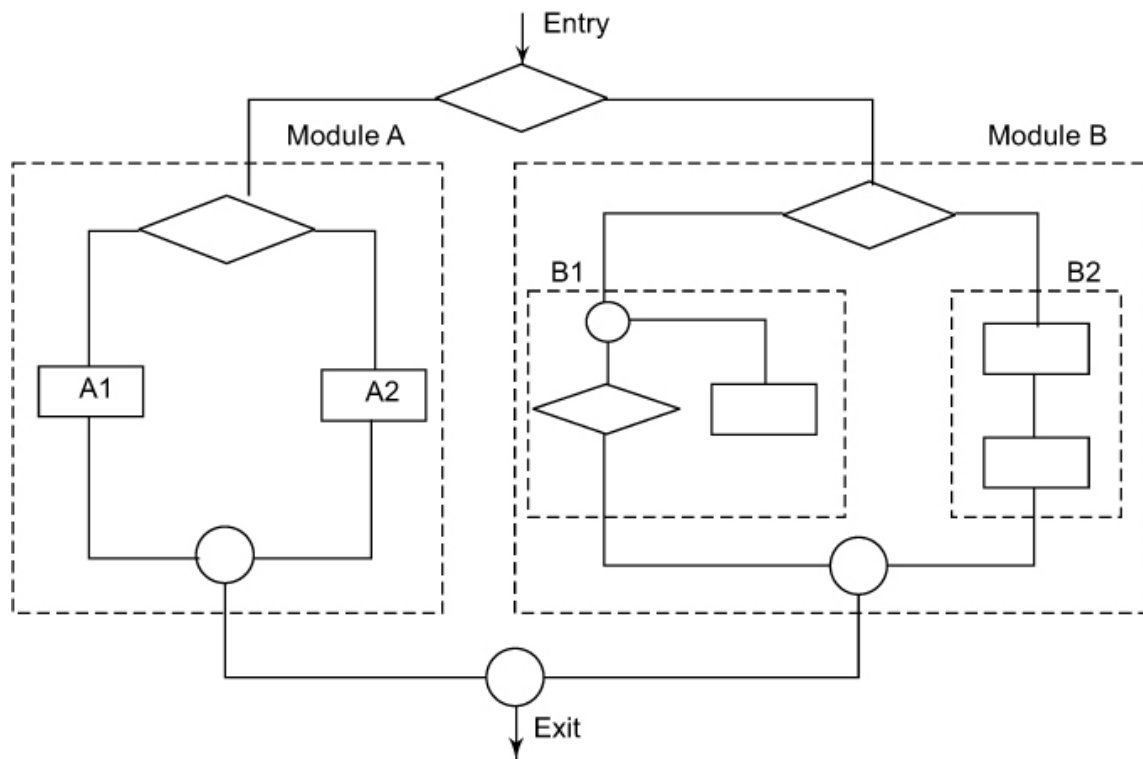
Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in [Fig. 3.6](#). This shows that C++ combines the power of structured programming with the object-oriented paradigm.



(a) First level of abstraction



(b) Second level of abstraction



(c) Detailed flow chart

Fig. 3.5 *Different levels of abstraction*

The if statement

The **if** statement is implemented in two forms:

- Simple **if** statement
- **if...else** statement

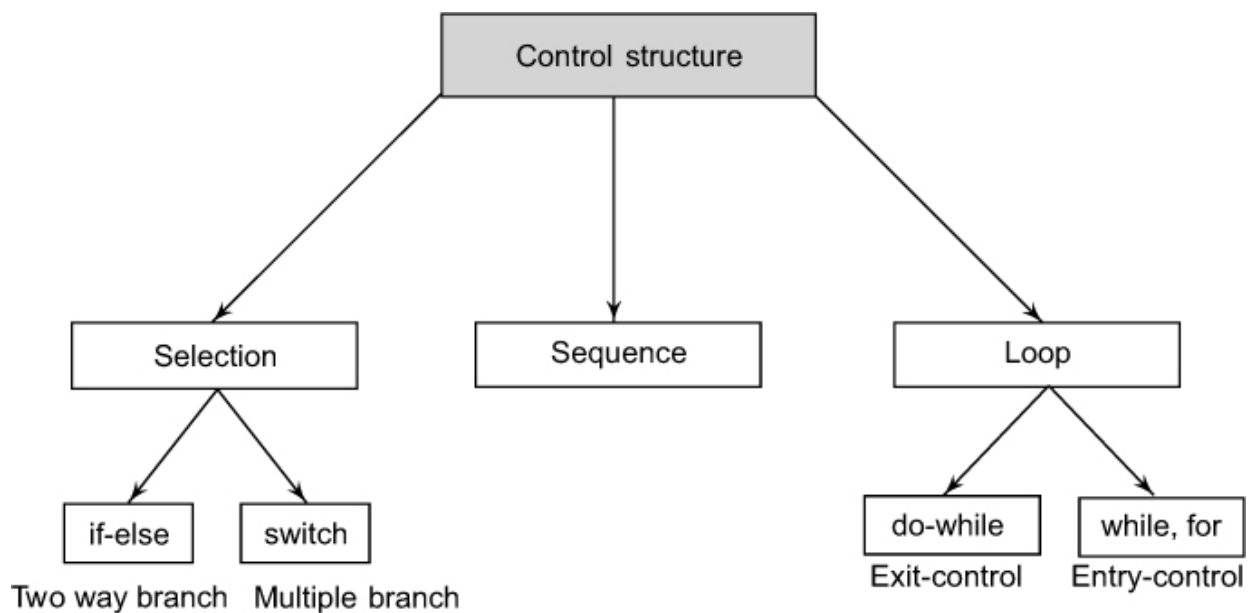


Fig. 3.6 *C++ statements to implement in two forms*

Examples:

Form 1

```
if(expression is true)
{
    action1;
}
action2;
action3;
```

Form 2

```
if(expression is true)
{
    action1;
}
else
{
    action2;
}
action3;
```

The switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
{
    case1:
    {
        action1;
    }

    case2:
    {
        action2;
    }
    case3:
    {
        action3;
    }
    default:
    {
        action4;
    }
}
action5;
```

The do-while statement

The **do-while** is an *exit-controlled* loop. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
    action1;
}
while(condition is true);
action2;
```

The while statement

This is also a loop structure, but is an *entry-controlled* one. The syntax is as follows:

```
while(condition is true)
{
    action1;
}
action2;
```

The for statement

The **for** is an *entry-controlled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
    action1;
}
action2;
```

The syntax of the control statements in C++ is very much similar to that of C and therefore they are implemented as and when they are

required.

Summary

- ☐ C++ provides various types of tokens that include keywords, identifiers, constants, strings, and operators.
- ☐ Identifiers refer to the names of variables, functions, arrays, classes, etc.
- ☐ C++ provides an additional use of **void**, for declaration of generic pointers.
- ☐ The enumerated data types differ slightly in C++. The tag names of the enumerated data types become new type names. That is, we can declare new variables using these tag names.
- ☐ In C++, the size of character array should be one larger than the number of characters in the string.
- ☐ C++ adds the concept of constant pointer and pointer to constant. In case of constant pointer, we can not modify the address that the pointer is initialized to. In case of pointer to a constant, contents of what it points to cannot be changed.
- ☐ Pointers are widely used in C++ for memory management and to achieve polymorphism.
- ☐ C++ provides a qualifier called **const** to declare named constants which are just like variables except that their values can not be changed. A **const** modifier defaults to an **int**.
- ☐ C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is type casting.

- ☐ C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- ☐ A reference variable provides an alternative name for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value of the other variable.
- ☐ A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.
- ☐ A major application of the scope resolution (::) operator is in the classes to identify the class to which a member function belongs.
- ☐ In addition to **malloc()**, **calloc()** and **free()** functions, C++ also provides two unary operators, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.
- ☐ C++ also provides manipulators to format the data display. The most commonly used manipulators are **endl** and **setw**.
- ☐ C++ supports seven types of expressions. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.
- ☐ C++ also permits explicit type conversion of variables and expressions using the type cast operators.
- ☐ Like C, C++ also supports the three basic control structures namely, sequence, selection and loop, and implements them using various control statements such as, **if**, **if...else**, **switch**, **do..while**, **while** and **for**.

Key Terms

array | associativity | automatic conversion | backslash character | bitwise expression | **bool** | boolean expression | branching | call by reference | **calloc()** | character constant | chained assignment | **class** | compound assignment | compound expression | **const** | constant | constant expression | control structure | data types | decimal integer | declaration | **delete** | dereferencing | derived-type | **do... while** | embedded assignment | **endl** | entry control | enumeration | exit control | explicit conversion | expression | float expression | floating point integers | **for** | formatting | free store | **free()** | function | hexadecimal integer | identifier | **if** | **if... else** | implicit conversion | initialization | integer constant | integral expression | integral widening | iteration | keyword | literal | logical expression | loop | loop structure | **malloc()** | manipulator | memory | named constant | **new** | octal integer | operator | operator keywords | operator overloading | operator precedence | pointer | pointer expression | pointer variable | reference | reference variable | relational expression | repetition | scope resolution | selection | selection structure | sequence | sequence structure | **setw** | short-hand assignment | **sizeof()** | straight line | **string** | string constant | **struct** | structure | structured programming | switch | symbolic constant | token | type casting | type compatibility | **typedef** | **union** | user-defined type | variable | **void** | water-fall model | **wchar_t** | **while** | wide-character

Review Questions

- 3.1 Enumerate the rules of naming variables in C++. How do they differ from ANSI C rules?
- 3.2 An **unsigned int** can be twice as large as the **signed int**. Explain how?
- 3.3 Why does C++ have type modifiers?

- 3.4** What are the applications of **void** data type in C++?
- 3.5** Can we assign a **void** pointer to an **int** type pointer? If not, why? How can we achieve this?
- 3.6** Describe, with examples, the uses of enumeration data types.
- 3.7** Describe the differences in the implementation of **enum** data type in ANSI C and C++.
- 3.8** Why is an array called a derived data type?
- 3.9** The size of a **char** array that is declared to store a string should be one larger than the number of characters in the string. Why?
- 3.10** The **const** was taken from C++ and incorporated in ANSI C, although quite differently. Explain.
- 3.11** How does a constant defined by **const** differ from the constant defined by the preprocessor statement **#define**?
- 3.12** In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?
- 3.13** What do you mean by dynamic initialization of a variable? Give an example.
- 3.14** What is a reference variable? What is its major use?
- 3.15** List at least four new operators added by C++ which aid OOP.
- 3.16** What is the application of the scope resolution operator **::** in C++?
- 3.17** What are the advantages of using **new** operator as compared to the function **malloc()**?
- 3.18** Which manipulator is used to control the precision of floating point numbers? Support your answer with the help of an example.
- 3.19** Illustrate with an example, how the **setw** manipulator works.

3.20 How do the following statements differ?

(a) `char * const p;`

(b) `char const *p;`

Debugging Exercises

3.1 What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
    int i=0;
    i=400*400/400;
    cout << i;
}
```

3.2 Identify the error in the following program.

```
#include <iostream.h>

void main()
{
    int num[]={1,2,3,4,5, 6};
    num[1]= =[1]num ? cout<<"Success" : cout<<"Error";
}
```

3.3 Identify the errors in the following program.

```
#include <iostream.h>

void main()
{
    int i=5;
    while(i)
    {
```

```

        switch(i)
        {
        default:
        case 4:
        case 5:

        break;
        case 1:
        continue;

        case 2:
        case 3:
        break;

        }
        i--;
    }
}

```

3.4 Identify the error in the following program.

```

#include <iostream.h>
#define pi 3.14

int squareArea(int &);
int circleArea(int &);

void main()
{
    int a=10;
    cout << squareArea(a) << " ";
    cout << circleArea(a) << " ";
    cout << a << endl;
}
int squareArea(int &a)
{
    return a * = a;
}

```

```
int circleArea(int &r)
{
    return r = pi * r * r;
}
```

3.5 Identify the error in the following program.

```
#include <iostream.h>
#include <malloc.h>

char* allocateMemory();

void main()
{
    char* str;
    str = allocateMemory();
    cout << str;
    delete str;
    str = " ";
    cout << str;
}


char* allocateMemory()
{
    str = "Memory allocation test, ";
    return str;
}
```

3.6 Find errors, if any, in the following C++ statements.

- (a) long float x;
- (b) char *cp = vp; // vp is a void pointer
- (c) int code = three; // three is an enumerator
- (d) int *p = new; // allocate memory with new
- (e) enum (green, yellow, red);

- (f) `int const *p = total;`
- (g) `const int array_size;`
- (h) `for (i=1; i<10; cout << i << "\n";`
- (i) `int & number = 100;`
- (j) `float *p = new int [10];`
- (k) `int public = 1000;`
- (l) `char name[3] = "USA";`

Programming Exercises

- 3.1 Write a function using reference variables as arguments to swap the values of a pair of integers. 
- 3.2 Write a function that creates a vector of user-given size **M** using **new** operator.
- 3.3 Write a program to print the following output using **for** loops.

```
1
22
333 4444
55555
.....
```

- 3.4 Write a program to evaluate the following investment equation

$$V = P(1 + r)^n$$

and print the tables which would give the value of V for various combination of the following values of P , r and n :

P : 1000, 2000, 3000..... 10,000

r : 0.10, 0.11, 0.12.....0.20

n : 1, 2, 3.....10


Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$$V = P(1 + r)$$

$$P = V$$

In other words, the value of money at the end of the first year becomes the principal amount for the next year, and so on.



3.5 An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and **count** the votes cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot', and the program should also count the number of spoilt ballots. 

3.6 A cricket team has the following table of batting figures for a series of test matches:

<i>Player's name</i>	<i>Runs</i>	<i>Innings</i>	<i>Times not out</i>
Sachin	8430	230	18
Saurav	4200	130	9
Rahul	3350	105	11
.	.	.	.
.	.	.	.

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

3.7 Write programs to evaluate the following functions to 0.0001 % accuracy.

(a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$

(b) $\text{SUM} = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \dots$

(c) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$

3.8 Write a program to print a table of values of the function $y = e^{-x}$ for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

Table for Y = EXP [-X]

X	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.0									
1.0									
.									
.									
.									
9.0									

3.9 Write a program to calculate the variance and standard deviation of N numbers.

$$\text{Variance} = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

where $\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$

3.10 An electricity board charges the following rates to domestic users to discourage large consumption of energy:

For the first 100 units - 60P per unit

For next 200 units - 80P per unit

Beyond 300 units - 90P per unit

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added. Write a program to read the names of users and number of units consumed and print out the charges with names. 